Object Evolution:
Adding Runtime Class Mutability to the JVM

by

Richard C. Schalck

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 2005

Author .....
Department of Electrical Engineering and Computer Science
10 January 2005

Certified by .....
Barbara Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Object Evolution:

Adding Runtime Class Mutability to the JVM

by

Richard C. Schalck

Submitted to the Department of Electrical Engineering and Computer Science
on 10 January 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis proposes an extension to the Java programming language in which a program may specify that an object can change its class at runtime. The project proposes a programmer interface for using this new capability. Given the proposed interface, a particular strategy for implementing the interface is presented. The JVM portion of this strategy is detailed and implemented on top of an existing JVM. The modified JVM is tested using scenarios designed to verify the new runtime class mutability capability.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering

Acknowledgments

My studies at MIT, including this thesis, have been supported by the Raytheon Advanced Study Scholarship Program. As such, I would like to thank Paul Weeks, Anthony Pellegrino, Joe McDougall, and Jeff Lewitzky for supporting my application to this program. A special thanks to Nina Caissie and Joe McDougall for helping with the transitions between work and school.

I would also like to thank the DD(X) SITB Team at Raytheon for serving as a great review committee for my ideas, especially Dan Rasmussen and Ryan McLoughlin.

I will always be indebted to the JikesRVM Project and its contributors. This thesis benefited immensely from the quality of the documentation, the quality of the code commenting, and the straightforward implementation of JikesRVM.

I am very grateful to Barbara Liskov for her guidance on this project. Her questions always pointed me to the areas that needed more thought and elaboration.

Finally, I would like to express my gratitude to my family for always supporting me in all of my academic work.

Contents

Chapter 1

Motivation

Many problem domains include mutable objects that change over time. Based on the interactions of the objects in the domain, the instance variables (or fields) of these objects are modified. However, in some domains, there are objects that change not only the values of the fields, but also the number and type of fields that the object carries. It may even be the case that the behavior of these objects (i.e. the object's methods) may need to change over time. This situation is apparent in a domain that represents a group of humans and their interactions. In this domain, one might imagine many instances of a Human class. This class may have instance variables that include age, weight, etc. The values of these fields would likely change over time. In addition, as the Human became an adult, new instance variables might be desirable, like occupation. It would also be helpful if the meaning of the methods of Humans also progressed with time. For example, one might expect the speaking behavior of an object of class Human to become more refined as time progressed from infancy to childhood to adulthood.

In moving these problem domain representations to a Java implementation some difficulties are encountered. Objects in which additional instance variables may be added and methods may be replaced would be ideal. However, since Java is class-based as opposed to prototype-based, the objects are constrained to conform to a class definition. Therefore, the next best option would be to allow the object to change classes at runtime. Such a capability, however, does not exist in *The Java Language*

Specification or *The Java Virtual Machine Specification* [4, 10].

One might suggest that to accomplish the desired runtime class swapping effect in Java, simply create a new object of the new class, copy over any state that is to be preserved, and discard the old object. There are a couple of problems with this solution. First, there may be other references to the old object that never get updated to the new object. Second, if a lot of state is to be preserved, the copying operation may be expensive.

Typically, in Java, the strategy design pattern or the state design pattern is used when an object must change instance variables or methods over time [3]. In the strategy design pattern, the above domain might be implemented by having a HumanShell class with an instance variable of class type HumanStrategy. There would then be classes InfantStrategy, ChildStrategy, and AdultStrategy, all subclasses of HumanStrategy with different instance variables and method behavior. As an object of class HumanShell progressed over time, the various HumanStrategy subclasses could be swapped into the instance variable to simulate changes to the number and type of instance variables and the behavior of methods of the Human class in the problem domain.

Although the design pattern solutions can actually model the problem domain, the implementation details do not closely follow what is perceived to be happening in the problem domain. Ideally, the Java implementation would have Infant, Child, and Adult subclasses of an abstract Human class. Then, a variable of class type Human could start out as an object of class Infant. That object could then evolve into a Child, and finally, evolve into an Adult. Code that used the evolving object could use the standard interface established by the Human superclass. As the object evolved, the method behavior of the object would change as the methods of different classes would be invoked as appropriate. In addition, instance variables might be added or removed to the object as the class of the object changed.

In addition to mapping poorly to the problem domain, the design pattern approach may also suffer from the requirement that everyone working on the implementation understand and follow the design rules for implementing the pattern. If instead, Java

had built-in features to allow an object to change its class during runtime (i.e. evolve), then the mechanism would be standardized across the language as opposed to relying on standardization within a project.

Additional examples of scenarios where runtime class mutability might be of particular value include [12]:

- **Construction of a structure of objects before classes are known**

  Consider a large network of computers as the problem domain. It may be necessary to first establish the interconnections between the computers before knowing the particular platform details. In this situation, the implementation would consist of a Computer class with several direct subclasses like CiscoComputer, LinuxComputer, SolarisComputer, and WindowsComputer. First, the topology of the network could be constructed with objects of class Computer. Then, as the platform of each computer was determined, the objects could evolve into the appropriate subclass.

- **The characterization of an object may be refined over time**

  Consider a radar target tracking system as the problem domain. In this implementation, the information known about an object gradually improves over time. For instance, when the radar first detects a target, an object of class Target is instantiated. Then, as more data is collected, the object evolves into an AirTarget, which is a subclass of Target. Finally, additional data triggers the evolution of the object to a MissileTarget, which is a subclass of AirTarget.

- **Cheap proxies for expensive full-blown objects**

  Consider a problem domain in which full-blown objects are very expensive to create and may never actually be used. In this implementation, proxy objects of the CheapProxyModel class are first instantiated. Then, as it is determined that a particular object will be used, it evolves into an ExpensiveRealModel, which is a subclass of the CheapProxyModel class.

- **Hot swapping classes in a running system**

  With an evolution capability, the class of an existing object could be "upgraded" by having the object evolve into a new class, which would incorporate any fixes or new capabilities. All other objects that rely on the evolved object would continue to see the same interface. The JVM would be able to perform this "upgrade" on any object.

These examples demonstrate the potential benefit of an evolution capability in Java. The next chapter sets out to explore how other languages have included this capability.

Chapter 2

Other Languages

Runtime class mutability is not a new concept. Other languages have implemented this capability directly or implemented functionality that enables this capability. These languages tend to be less formal about type issues, relying more on the user for type checking. Two important examples are Smalltalk and the Common LISP Object System (CLOS).

2.1 Smalltalk

In Smalltalk, runtime class mutability may be accomplished using the strategy of creating a new object of the new class, copying over any state that is to be preserved, and then updating all references to the old object so that they point to the new object. In Java, updating all references to a particular object is usually not possible. However, Smalltalk provides a built-in primitive that provides this functionality, the `becomes:` meta operation. The `becomes:` operation has the following format:

```
anObject becomes: anotherObject
```

In this operation, `anObject` and `anotherObject` are variables pointing to any defined objects (possibly the same object). There need not be any particular relationship between the classes of the two objects referenced by the variables. The effect of the meta operation is to move **all** references to the object referenced by `anObject`

to the object referenced by `anotherObject` and to move **all** references to the object referenced by `anotherObject` to the object referenced by `anObject`. Therefore, if before the operation, the `anObject` value is of class Monkey and the `anotherObject` value is of class Horse, then after the operation, the `anObject` value will be of class Horse and the `anotherObject` value will be of class Monkey. The `becomes:` operation updates all references to these two objects, so it is as if the two objects have been swapped everywhere.

Using the `becomes:` operation to implement runtime class mutability in Smalltalk requires the programmer to ensure a proper relationship between the original class of the object and the new class. The programmer must ensure that the new class will understand the messages passed to the object after the evolution (i.e. only valid methods of the new class are invoked). Of course, in Smalltalk, this sort of type checking is normally the responsibility of the programmer [9].

2.2 CLOS

CLOS implements runtime class mutability directly through the inclusion of a built-in change-class generic function. The change-class generic function takes as arguments the instance to be changed and the name of the new class to which the instance is to evolve. CLOS does not place any constraints on the relationship between the current class of the object and the new class. CLOS defines a set of rules for determining how the values of the slots of the object will be modified in light of the slots defined by the old class and the new class. These rules are mainly governed by the names of the slots. The change-class function also provides means for the programmer to customize how the slot values are transformed and initialized during the evolution.

CLOS is able to change the class of the instance "in place." In other words, the programmer need not worry about updating other references because the implementation updates the class of the instance without moving it in memory. The implementation of this is an important example to be leveraged in adding the evolution capability to Java.

Again in CLOS, as in the Smalltalk scheme, the programmer must ensure that any generic functions that use the object have a defined method with a specialized parameter that matches the new class of the object. In other words, once the object has changed classes, the programmer must be sure to only call methods of the new class (or its superclasses) on the new object [8].

Both Smalltalk and CLOS provide important examples of how runtime class mutability can be presented to the programmer. Their respective implementations also provide great insight into how this same capability can be integrated into Java. These examples are leveraged in the design and implementation presented next.

Chapter 3

Object Evolution

This chapter describes the proposed programmer interface for a Java runtime class mutability capability (i.e. an evolution capability), and the design to implement it.

3.1 Programmer Interface

CLOS provides an excellent example of how an evolution capability might be presented to the Java programmer. From the programmer's perspective, the CLOS change-class generic function does exactly what is desired - it changes the class of an object to the new class without moving the object. The programmer does not have to worry about updating references because the object is exactly where it was before the change. In addition, the change-class function provides a means for customizing the updating and initializing of the slot values of the object as a result of the class change [8].

Closely following the change-class example, this thesis proposes that evolution be incorporated into the Java language through a new keyword, `evolveto`, and a new type of expression, an evolution expression:

ReferenceVariable `evolveto` *ClassType*`;`

After the execution of this expression, the object referred to by *ReferenceVariable* is of class *ClassType*. This expression is exactly what is desired in Java, but there are a few issues that must be addressed.

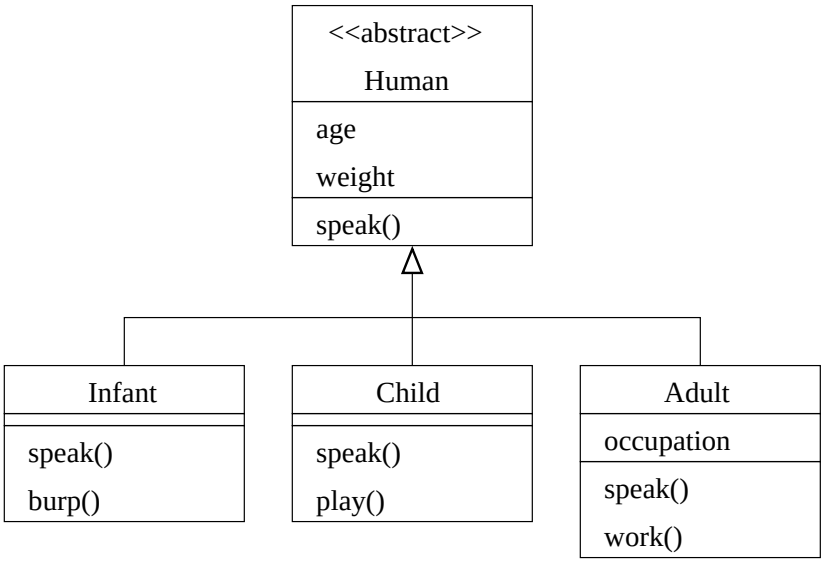

Figure 3-1: Human Class Hierarchy

3.1.1 Type Safety

In translating from the CLOS change-class generic function to a Java version of the capability, the principle concern is maintaining Java type safety. For the purpose of this thesis, type safety means that the value of any variable must conform to an interface defined by the type of that variable. The type of the value of a variable must be compatible with the declared type of the variable.

An obvious consequence of type safety is that evolution expressions must maintain the type correctness of the involved variable. Referring back to the prototype of evolution expressions, an instance of class *ClassType* must be assignment compatible with the declared type of *ReferenceVariable*. Assignment compatibility is clearly laid out in *The Java Language Specification*. Java compilers perform assignment compatibility checks in a variety of expressions, so a compiler could also perform this check on evolution expressions.

Although assignment compatibility in evolution expressions is necessary, it is not sufficient to ensure universal type safety. The next example demonstrates how a type violation might arise without additional constraints on evolution.

The class diagram depicted in Figure 3-1 shows a Java implementation of the humans problem domain presented in the last chapter. The Infant, Child, and Adult

classes are all subclasses of the Human abstract class. Therefore, an instance of the Infant, Child, or Adult classes can be the value of a variable of class type Human. Given this class structure, consider the following Java code segment.

```
Infant an_infant = new Infant();
Human a_human = an_infant;
a_human.speak();                    // Infant speak() invoked

a_human evolveto Child;
a_human.speak();                    // Child speak() invoked

a_human evolveto Adult;
a_human.speak();                    // Adult speak() invoked

an_infant.burp();                   // method not found!!!
```

This code sequence demonstrates how the methods of an object might change as an object evolves from class to class, which is exactly what is desired as a model of the problem domain. However, the `an_infant` variable violates type safety. Both the `a_human` and `an_infant` variables point to the same object. When the `a_human` object evolved into a Child, the Child object no longer satisfied the type of the `an_infant` variable. When the `burp()` method of the Infant class is invoked on the `an_infant` variable, the type safety violation is revealed.

To maintain type safety, the class type specified in the evolution expression must be assignment compatible with any variable that references the evolving object. Appealing to the Liskov Substitution Principle [11], this requirement is equivalent to saying that the new class type must be a subtype of all the variables that refer to the object. The problem is that the set of variable types that refer to an object is generally not known at compile time. Even at runtime, computing this set of types is normally not possible. However, it is known that the class type associated with the object's pre-evolution class is a subtype of all variables that refer to the object (otherwise, assignment compatibility would have precluded a particular variable from referring to the object). Therefore, if the new class is restricted to be a subclass of the pre-evolution class of the object, then the requirement for all variables will be

satisfied by the transitivity of subtyping. This restriction will be referred to as the subclass rule.

The subclass rule can be simply stated as follows: The new class specified in an evolution expression must be a subclass of the current class of the evolving object. Although this is a fairly simple rule, there is still one additional complication - the class of an object referred to by a variable is not necessarily known until runtime. The evolution expression cannot be allowed to complete normally if the runtime check of the subclass rule fails (because otherwise type safety might be violated). Based solely on simplicity, this thesis proposes that if the runtime check of the subclass rule fails, an EvolveException, a subclass of RuntimeException, is raised. With this approach, if the programmer chooses, the EvolveException may be caught and handled appropriately.

Given the subclass rule, it is prudent to re-examine the possible applications of evolution. As already demonstrated, an evolution capability is not compatible with a Java implementation of the humans problem domain in which the Infant, Child, and Adult classes are siblings and all direct subclasses of the Human class. This translation of the problem domain into an implementation relies on the "is a" interpretation of inheritance. However, if an alternative interpretation is used, then the humans problem domain may still be a candidate for evolution. For instance, if inheritance means that a subclass can do everything its direct superclass can do plus possibly more, then the implementation would result in the class hierarchy depicted in Figure 3-2. In this class hierarchy, the code presented earlier would satisfy the subclass rule, maintain type safety, and still provide a possibly useful simulation of the problem domain.

Examining each of the other scenarios pointed out in Chapter 1 reveals that the subclass rule does not diminish the usefulness of the evolution capability, although care may be necessary in constructing the best class hierarchy to utilize evolution.

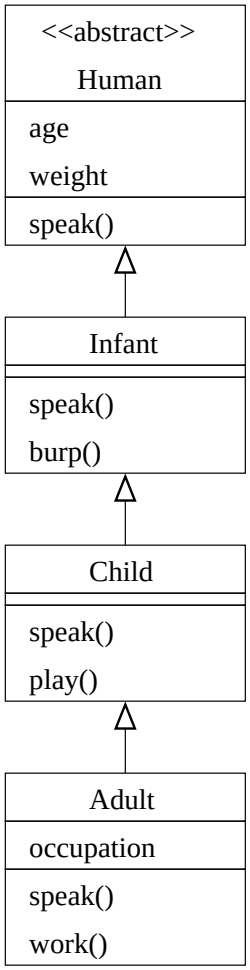

Figure 3-2: Alternate Human Class Hierarchy

3.1.2 Field Initialization

CLOS has established a set of rules for determining how slots (fields in Java) are to be maintained, initialized, or discarded when the change-class generic function is executed. In addition, special provisions are available to allow the programmer to customize this process. Comparing Java to CLOS, the subclass rule simplifies considerably how fields might be handled during evolution.

The subclass rule means that an evolution can result in either no change to the number of fields or the addition of new fields. Obviously, the pre-evolution field values of the object should remain unchanged through evolution thus preserving the existing state of the object. The question remains as to how fields introduced due to the new class will be initialized. One approach would be to provide an explicit mechanism where the programmer could place evolution initialization code. For instance, the evolution protocol could be expanded to include the invocation of a special `<evolve>` method of the new class. The programmer could then place evolution initialization code in this method if needed. This `<evolve>` method would then serve the same purpose as the constructor (or `<init>` method) in regular instance creation. One could also imagine chaining of the `<evolve>` methods if the new evolution class is not a direct subclass of the pre-evolution class of the object. This chaining would be quite similar to the chaining of `super()` invocations in regular instance creation.

Although the `<evolve>` method approach may be appealing, for simplicity, this thesis will take a different approach. From the programmer's perspective, evolution will have the following initialization details for new fields added because of evolution. If the new class does not have a no-argument constructor, then the new fields will be initialized to the default Java values (e.g. `null` for object references, `0` for integers, etc.) [4]. Otherwise, imagine a new instance of the new evolution class is created in the usual Java way using the `new` keyword and the no-argument constructor. Consequently, all fields are initialized in the normal way. Next, for each new field due to evolution, the value found in the newly instantiated instance is used as the evolving object's initial field value. Note that this simplification does have ramifications for

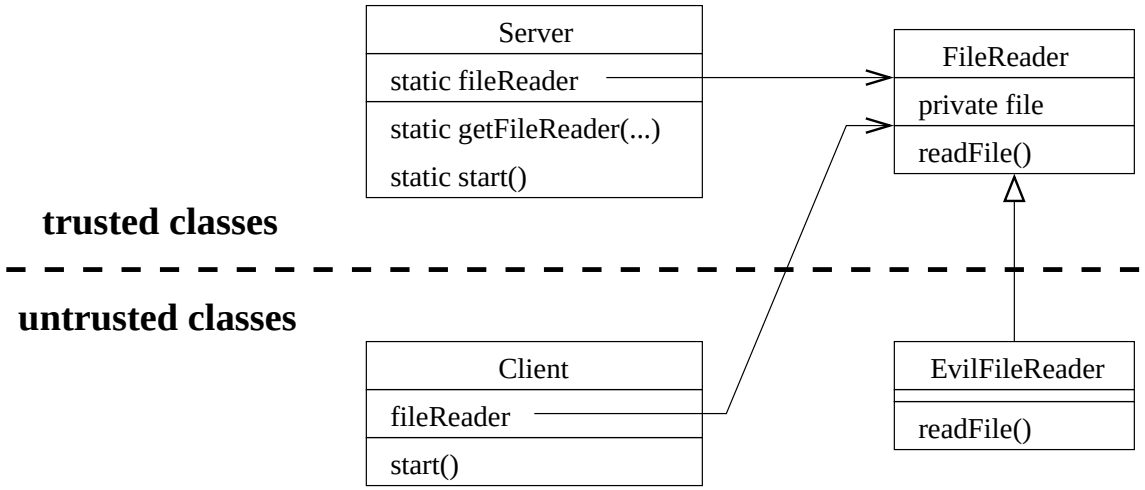

Figure 3-3: Trusted and Untrusted Classes

the initialization of new fields where the normal initialization depends on the values of fields already defined in the pre-evolution class of the object. Instead of using the field values that exist in the evolving object, the values that would exist in a new instance are used. Later it will be clear that this decision is influenced primarily by implementation details.

3.1.3 Controlling Evolution

In some circumstances, an evolution capability may be too powerful to allow without any checks or controls. This is especially true when untrusted classes (e.g. from an outside vendor) may be present.

Consider an example where the running program consists of a set of trusted classes and a set of untrusted classes. The behavior of the trusted classes is well understood and known to conform to certain representation invariants. The behavior of the untrusted classes has not been verified and could even be malicious.

Figure 3-3 shows an example class diagram involving trusted and untrusted classes. Suppose that the static `getFileReader()` method of the Server class is invoked by a Client object to obtain a reference to the same object held by the static `fileReader` field of the Server class. In standard Java, this would be no cause for concern; the trusted Server class can be sure that the only thing the Client object can do to

the object is call its `readFile()` method. Since the `readFile()` method does not modify the FileReader object in any way, the Server class can be secure in continuing to use the FileReader object, invoking the object's `readFile` method when needed. However, with an evolution capability available, the Client object can do much more to the FileReader object than just invoking its `readFile()` method. The Client object could use an evolution expression to change the class of the FileReader object to the EvilFileReader class, an untrusted class that is defined to be a subclass of the FileReader class. The EvilFileReader class overrides the `readFile()` method of the FileReader class. Now, when the Server class invokes the `readFile()` method of its `fileReader` variable, instead of getting the result of the FileReader `readFile()` method, the EvilFileReader `readFile()` method returns whatever malicious data it likes. The effect of the malicious data on the Server class is potentially dangerous.

In situations where untrusted classes are present, a way to limit the evolution capability is required to protect against the above scenario. Following the Serializable and Cloneable examples in standard Java, a Java interface can be used to identify the classes for which evolution should be allowed or disallowed. In an environment with untrusted classes, it is safest to assume that objects of all classes should be disallowed from evolution and then identify those classes for which it is safe. These classes would implement the Evolvable interface, which does not actually specify any methods. By segregating Evolvable objects from untrusted classes, the attack presented above can be avoided. If necessary, an Evolvable object could be encapsulated as a private field of a non-Evolvable class and then safely passed to an untrusted class.

As with the subclass rule, whether the class of the object in an evolution expression implements the Evolvable interface is, in general, not known until runtime. Before the runtime check of the subclass rule, a check is performed to ensure that the class of the object implements the Evolvable interface. If the check fails, then an EvolveException is raised. If the check succeeds, then the subclass rule is checked as before.

Given this protection scheme, it is again useful to re-examine if there is an impact on the possible applications of evolution. The introduction of the Evolvable interface requires that there be some anticipation of evolution in the class hierarchy design.

Where the problem domain exhibits class changing behavior, evolution can definitely be anticipated during the class hierarchy design. However, with the hot-swapping of classes example described in Chapter 1, the advantage of the evolution capability is that any object can be upgraded, without prior planning. If there is an anticipation that a class will eventually need to be upgraded, it would be superior to integrate the upgrading capability into the design through the use of interfaces and other mechanisms as opposed to relying on an evolution capability.

If the classes in a particular implementation are all trusted, it would be useful to have the option to allow the evolution capability to operate without the Evolvable check. This would make the hot-swapping upgrades applicable again. Although this thesis will not include this option in its specification of the evolution capability, the implementation presented in Section 3.2 could easily admit this feature as a JVM flag.

3.2 Design and Implementation

3.2.1 Java, Java Compiler, and JVM

The programmer interface detailed in Section 3.1 is only a specification of how evolution expressions in the Java language translate to program behavior. In the path from Java language to program behavior, there is more than one place where the evolution implementation can be inserted.

Initially, some time was spent examining whether the evolution capability could be implemented as a source-to-source translator to be applied before the Java compiler. In the source-to-source solution, all references to all objects would have to be rerouted through an object table to facilitate the universal replacement of an object with a new object of a new class. There are a few complications with this approach. First, correctly and completely replacing all object references with some sort of proxy that points to an object table might be quite challenging considering the variety of ways in which an object reference may occur in Java. Second, library packages

that take in object references would also have to be run through the source-to-source translator and recompiled. Finally, routing all object references through an object table could be a costly overhead to accomplish the occasional class changing operation [9]. Considering these issues, the source-to-source translator does not provide the best solution.

Implementing the evolution capability within the JVM is the most natural solution as this is where the runtime representation of an object exists. This strategy requires the introduction of a new bytecode opcode that is generated by the Java compiler and interpreted by the JVM.

For the Java compiler's part, this JVM strategy entails translating the newly introduced Java `evolveto` operator to a new *evolve* JVM opcode. The format of the new *evolve* opcode is similar to the *instanceof* opcode; it is followed by two bytes used to calculate an offset into the current constant pool where a class descriptor is located. For the *evolve* opcode, this location in the constant pool indicates the new class of the object. The Java compiler also has to perform the assignment compatibility check on the new class as discussed earlier. These changes to the Java compiler are relatively simple. The Java compiler mechanism used to handle the `instanceof` operator can be reused for handling `evolveto` expressions. The assignment compatibility check for the `evolveto` expression can be borrowed from a variety of other places in the Java compiler where this check is required.

Because of the simplicity of the changes, this thesis project will not set out to modify a Java compiler to perform the `evolveto` expression compilation. Instead, in order to test the modified JVM, `instanceof` expressions are used in Java code to mark places where an `evolveto` expression should occur. The `instanceof` expressions use the class name that is to serve as the new evolution class. This Java code is then compiled into bytecode using a standard Java compiler. Finally, using a class file reader and a binary data editor, the *instanceof* opcode is edited to the *evolve* opcode.

3.2.2 JVM Selection

The JVM must be able to interpret the new *evolve* opcode. The first step in accomplishing this was to select a JVM implementation from which to start. Several JVMs were examined. The JVMs were evaluated based upon licensing, platforms, complexity, and documentation. JikesRVM version 2.3.2 was finally selected based primarily on the quality of the documentation and the fact that a large portion of the JVM is written in Java [1]. The JikesRVM 2.3.2 source code has been installed and built on a RedHat 9 / AMD Athlon platform using the build procedures provided in the JikesRVM documentation [6]. The build configuration used throughout this thesis project results in a JikesRVM lacking most standard JVM optimizations (the "Base-Base" configuration, see JikesRVM documentation). Without these optimizations, the resulting JikesRVM, with or without evolution modifications, is not competitive with commercial JVMs. However, since demonstration of the evolution capability is the foremost goal of this thesis, this is of minor consequence.

Subsequent references in this thesis to the JikesRVM refer to the unmodified JikesRVM. References to the Evolution JVM refer to a modified JikesRVM with the evolution capability described herein added. The Evolution JVM can be obtained at `http://web.mit.edu/~rcory/www/thesis/`

3.2.3 Evolve Opcode

The first change made to the JikesRVM towards the Evolution JVM was adding the machinery necessary to read the new *evolve* opcode, along with the two subsequent bytes used to form the offset into the constant pool where the new class of the object is specified. To accomplish this, the existing JikesRVM code for the *instanceof* opcode was leveraged heavily. The impact to the JVM stack due to the *evolve* opcode also must be specified. Again like the *instanceof* opcode, the *evolve* opcode assumes the top word on the stack is an object reference. This is the object that will have its class changed. However, unlike the *instanceof* opcode, which replaces the object reference on the top of the stack with the integer result of the operation, the *evolve* opcode

will simply pop the object reference off the top of the stack.

Note that the slightly different JVM stack behaviors of the *instanceof* and *evolve* opcodes affect the class file editing procedure described in Section 3.2.1. In practice, this has meant that the opcode that normally pops the result of the fake *instanceof* opcode must be replaced by a *nop* opcode. This is acceptable because the `instanceof` expression in the original Java was only serving as a place holder.

3.2.4 Object Representation

The Java Virtual Machine Specification explicitly states that it does not mandate any particular internal structure for the representation of objects. However, the specification does note that some of Sun's JVM implementations represent an object as a pair of two pointers, one that points to the class of the object and another that points to a memory area where the object data lives [10]. The Symbolics CLOS implementation uses a similar object representation to facilitate the change-class functionality. In this CLOS implementation, the object data is normally adjacent to the pair of pointers. When the change-class function is executed, the class pointer is changed to point to the new class and a new object data area is allocated. The original object data pointer is changed to point to the new object data area. At the next garbage collection (GC) point, the object data is once again made adjacent to the pair of pointers that represent the object [2]. Unfortunately, from the perspective of implementing the Evolution JVM, JikesRVM did not follow the Sun JVM and Symbolics CLOS examples. The JikesRVM object representation consists of a pointer to the class information of the object with the object data adjacent to the class pointer. There is no object data pointer; the object data must remain adjacent to the class pointer for it to be found by all of the components of JikesRVM [1].

Figure 3-4 shows the JikesRVM object representation for an object with two fields[1]. Every object has two words of object header. The TIB (Type Information Block) Pointer is the object's class pointer. The Status Word of the object holds

[1]The JikesRVM is extremely configurable, so its object representation is also configurable. However, the representation shown is typical.

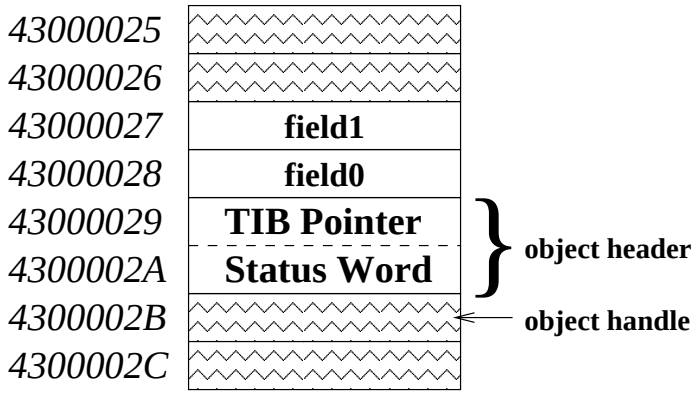

Figure 3-4: JikesRVM Object Representation

additional object state for locking and hash code functionality. As shown, the object's field data is simply adjacent to the object header. The object handle is the address stored in variables that point to this object.

Given the object representation of the JikesRVM, the Evolution JVM needs a scheme to handle changing an object to a class that requires additional object data space (this is the case whenever the new class introduces additional fields). Using Figure 3-4 as an example, the scheme must respect the possibility that another object may be using the memory space at address `0x43000026`. Therefore, the evolving object cannot simply grow to accommodate the new fields.

There are a few possible approaches to adding new fields to an object. One possibility is to allocate new space elsewhere and use the existing object data space to point to the new object data. Then, at the next GC, the object would be reconstituted to follow the original object representation. Another possible solution is to induce GC when an *evolve* opcode is encountered. During the shuffling of the GC process, the object data area of the object could be grown to accommodate the new data mandated by the new class. Yet another solution would be to change the object representation of JikesRVM to more closely follow the object data pointer example of the Sun JVM.

The approach chosen for the Evolution JVM closely follows the Symbolics CLOS example. Figure 3-5 shows the object representation in the Evolution JVM. The principle difference between the JikesRVM layout and the Evolution JVM layout is

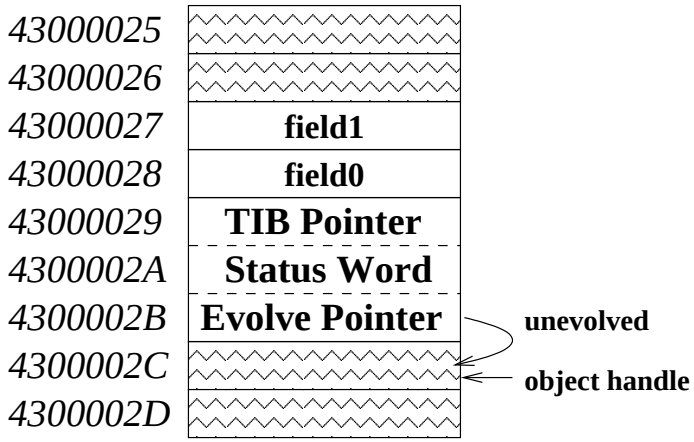

Figure 3-5: Evolution JVM Object Representation (unevolved object)

the addition of the Evolve Pointer in the Evolution JVM layout. The Evolve Pointer serves as the data area pointer for the object. Since the object in Figure 3-5 depicts an unevolved object, the Evolve Pointer points to the object itself. Note that the Evolve Pointer points to the same address that would be used to refer to this object (i.e. the object handle). In the Evolution JVM, whenever a new object is created, the Evolve Pointer of the new object is initialized to point to itself.

3.2.5 Runtime Stages and Transitions

The operation of the Evolution JVM can be divided into two major stages of operation with transitions between the stages. Figure 3-6 shows the stages and transitions of the Evolution JVM. The Unevolved Stage is almost identical to the standard JikesRVM operation. This stage exists before any *evolve* opcodes are encountered. When an *evolve* opcode is encountered while in the Unevolved Stage, the object being evolved is transformed and the operation of the Evolution JVM switches to the Evolved Stage. During the operation of the Evolved Stage, a GC may occur. This GC results in the reconstitution of all evolved objects so that they are indistinguishable from unevolved objects. After the GC, the operation of the Evolution JVM reverts to the Unevolved Stage.

Note that this model of the Evolution JVM operation assumes that GC is capable of reconstituting evolved objects. When an object evolves, additional space is allo-

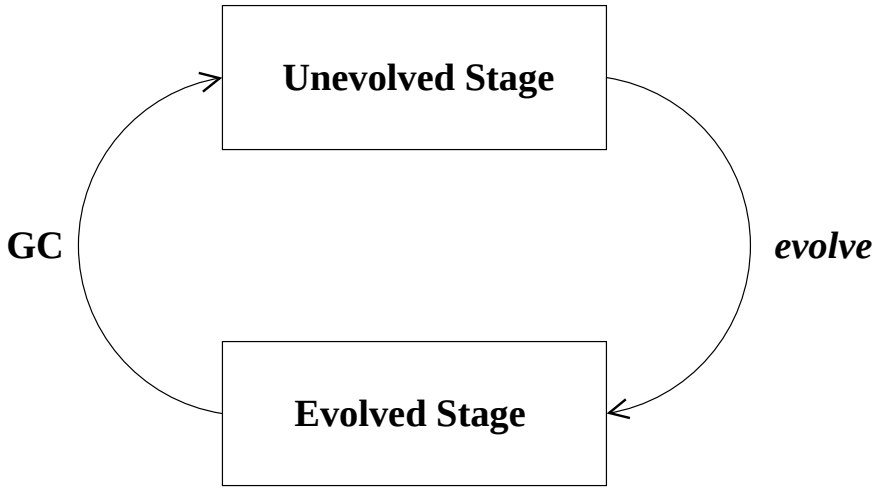

Figure 3-6: Stages of Evolution JVM

cated to accommodate additional fields added by the new class of the object. This additional space (which will be referred to as the annex) is normally not adjacent to the original space allocated for the object (which will be referred to as the original part). As already discussed, the reason for this is that other objects typically occupy the space surrounding the original space allocated for the evolving object. Reconstitution means that an evolved object is transformed so that the original part and the annex are merged into an unevolved object. For an unevolved object, all of the memory necessary to represent the object is contiguous, as in Figure 3-5. Therefore, for the GC to be able to reconstitute an evolved object, it must somehow be able to shuffle all of the currently defined objects to make space for the larger size of the object being reconstituted.

There are a variety of GC algorithms that are capable of shuffling objects during the GC process. This shuffling is possible as the GC algorithm copies the live objects from one area to another. Some of these GC algorithms perform copying during normal operation while others only perform copying as part of compacting of memory. The standard JikesRVM provides many different GC implementations from which to select. A Semispace Copying GC algorithm is used in the Evolution JVM implementation [1, 6]. This GC algorithm was selected because it

- performs the requisite copying that enables the shuffling of objects,
- performs copying as part of its normal operation instead of only for compacting,
- and is very simple and straightforward.

Although a Copying GC algorithm was selected for the Evolution JVM, there is no obvious reason why another algorithm, like Mark and Sweep or Reference Counting, would not work. In addition, a Generational approach to any of these algorithms would also be acceptable. It should be noted, however, that with some of these alternatives, a single GC cycle may not result in the reconstitution of all evolved objects. Therefore a single GC cycle might not result in the transition from the Evolved Stage to the Unevolved Stage. Since the Semispace Copying GC copies all live objects during a GC, the transition back to the Unevolved Stage is guaranteed with this algorithm.

Unevolved Stage

The operation of the Unevolved Stage is similar to the normal operation of the standard JikesRVM. The difference stems from the different object representations of the JikesRVM and the Evolution JVM. In the Evolution JVM, an extra step is required for field access, specifically *getfield* and *putfield* opcodes. For field access, the Evolve Pointer is first followed to find the location of the field data. Of course, in the Unevolved Stage, following the Evolve Pointer only leads back to the original object reference.

In the standard JikesRVM implementation, the *getfield* opcode results in the following machine instruction sequence on an IA32 architecture. `SP` is the stack pointer register; the object reference whose fields are being accessed is on the top of the stack. `T0` is a general purpose register. The value of `fieldOffset` is calculated when the method in which the *getfield* opcode appears is converted to machine instructions.

```
Mov_Reg_RegDisp(T0,SP,0)             // T0<-Mem[SP],
                                     //   Move object ref into T0
Mov_Reg_RegDisp(T0,T0,fieldOffset)   // T0<-Mem[T0+offset],
                                     //   field value loaded into T0
Mov_RegDisp_Reg(SP,0,T0)             // Mem[SP]<-T0, object ref popped,
                                     //   field value pushed
```

There are some modifications to this sequence for two word fields (e.g. double and long fields), but the basic idea is the same. The *putfield* opcode also follows similar logic.

The Evolution JVM adds a layer of indirection to the field access sequence using the Evolve Pointer. The basic scheme is to take the object reference, look at the Evolve Pointer of that object, and then move to the object pointed to by the Evolve Pointer. Field access then proceeds as usual. The following instruction sequence replaces the above sequence in the Evolution JVM. The `evolveOffset` is a global constant. The value of `evolveOffset` is -4 bytes, one word from the object handle (See Figure 3-5).

```
Mov_Reg_RegDisp(T0,SP,0)             // T0<-Mem[SP],
                                     //   Move object ref into T0
Mov_Reg_RegDisp(T0,T0,evolveOffset)  // T0<-Mem[T0+evolveOffset],
                                     //   Move to object data area
Mov_Reg_RegDisp(T0,T0,fieldOffset)   // T0<-Mem[T0+fieldOffset],
                                     //   field value loaded into T0
Mov_RegDisp_Reg(SP,0,T0)             // Mem[SP]<-T0,
                                     //   object ref popped,
                                     //   field value pushed
```

The utility of following the Evolve Pointer during the Unevolved Stage may not be immediately clear. In fact, it is not strictly necessary during the Unevolved Stage. The reason it is incorporated in the above code is that this same code can also be used during the Evolved Stage. This field access code works on both unevolved and evolved objects. Constructing the code in this way means that during normal operation, the Evolution JVM does not have to check whether an object is unevolved or evolved. This will become more clear as the Evolved Stage is explained.

Evolve Transition

When an *evolve* opcode is encountered, the Evolution JVM performs the following operations.

1. Ensure that the object on the top of the stack (the evolving object) is a not a null reference and implements the Evolvable interface. If either of these fail, an EvolveException is thrown.

2. Check to make sure that the new evolution class is a subclass of the current class of the evolving object. This is the verification of the subclass rule. If this check fails, an EvolveException is thrown.

3. Check if the new class adds additional fields compared to the current class of the evolving object. If not, skip to step 6. Otherwise, instantiate a new object of the new class. If there is a no-argument `<init>` method of the new class, apply this method to the new object. This new object will serve as the annex. The memory area allocated for the annex will generally be away from the evolving object's original area. Note that all of the fields of this new object (the annex) will be initialized using the standard initializations for the new class as specified in the no-argument constructor for the class (in accordance with Section 3.1.2).

4. Copy the contents of all of the fields in the original part into the fields of the annex.

5. Change the Evolve Pointer in the original part of the object to point to the object handle address of the new object created to serve as the annex.

6. Change the TIB Pointer in the original part of the object to point from its old class to the TIB of its new class.

Figure 3-7 shows the layout of an evolved object after the execution of an *evolve* opcode. The purpose of the annex, adding additional field space, is demonstrated by the new field. The Evolve Pointer in the original part of the object points to the

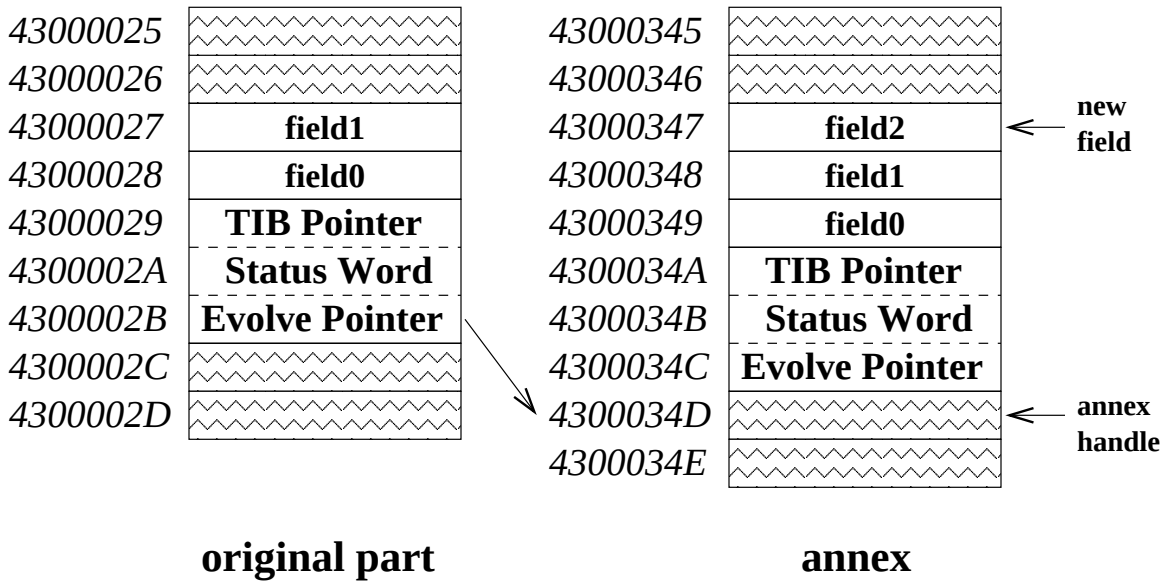

Figure 3-7: Representation of an Evolved Object

annex handle. Through this link, the field space of the annex can be used for the fields of the evolved object.

Evolved Stage

After an *evolve* opcode has been encountered, the Evolution JVM moves into the Evolved Stage of operation. In this stage, there are two types of object layouts. Unevolved objects conform to the layout of Figure 3-5, where the Evolve Pointer of an object points to itself. Of course, there are also evolved objects that follow the layout of Figure 3-7, where the Evolve Pointer points to an annex. To reduce the amount of code that has to handle these two different organizations, the original part of the evolved object is used as much as possible in the Evolved Stage. By original part, what is meant is the part of the object that exists before the *evolve* opcode is executed (in Figure 3-7, the address space from `0x43000027` to `0x4300002B`). In fact, it turns out that the only functionality that needs to use the annex is field access. Other object actions, like method invocation and locking, use the original part of the object, and therefore, no modifications of the JikesRVM are needed for this functionality in the Evolution JVM.

Method invocation relies solely on the TIB Pointer, which is updated to point

to the TIB of the new class during the execution of the *evolve* opcode. Method invocation does not follow the Evolve Pointer; it uses the TIB Pointer in the original part of the evolved object (address `0x43000029` in Figure 3-7). The annex is only used for field access. If the new class does not add fields to the evolving object, an annex is not even created - only the TIB Pointer is changed. The same policy is applied to the Status Word. The Status Word in the original part of the evolved object is always used. The Status Word provides state data for locking and hash code functionality.

If an evolved object is the target of a subsequent *evolve* opcode (which would only occur in the Evolved Stage), the steps are identical to those laid out above. A new object of the new class is instantiated, if necessary; this object will be called the new annex to distinguish it from the old, existing annex. The field values are then copied from the old annex to the new annex. Then the Evolve Pointer in the original part of the evolving object, **not** the Evolve Pointer in the old annex, is updated to point to the new annex. This effectively makes the old annex garbage, which will be collected in the next GC. This procedure always results in there being only one valid annex. There is no chaining of annexes, where the Evolve Pointer of one annex points to another annex and so on. Instead, if a new annex is needed, it replaces the old annex. The Evolve Pointer of an annex is never used (although it is initialized, as with all objects, to point to itself). This makes it possible to have an object evolve multiple times before an intervening GC.

GC Transition

As noted earlier, a Semispace Copying GC algorithm is used in the Evolution JVM. This algorithm divides the available memory into two equally sized areas, the FromSpace and the ToSpace. New objects are always allocated in the FromSpace. The invocation of a GC results in the following steps [1, 6, 7]:

1. The set of root pointers and origins is calculated. Root pointers can come from registers, the stack, global variables, and other locations. Each root pointer also

has an origin, or address. The origin is where the pointer is found. The origin may not actually be an address, like in the case of a register. These pointer and origin pairs are placed in a queue to be processed.

2. If the queue is empty, the GC is complete; stop. Otherwise, proceed to Step 3.

3. Remove the pointer/origin pair from the front of the queue.

   (a) If the pointer points into the ToSpace, then this pair can be ignored; Go to Step 2.

   (b) If the pointer points into the FromSpace, examine the location.

      i. If an object is at the location, copy the object to the end of the ToSpace. Overwrite the object in the FromSpace with a forward that points to its location in the ToSpace. Update the original pointer (at its origin) to point to the new location of the object in the ToSpace. Now, scan the moved object in the ToSpace. For each object reference in the moved object, add the pointer and its origin onto the end of the queue to be processed. Go to Step 2.

      ii. If a forward is at the location, update the original pointer (at its origin) to point to the target of the forward. Go to Step 2.

When the GC is complete the names of the two memory areas, FromSpace and ToSpace, are swapped.

There are only two modifications needed to this algorithm to accomplish the reconstitution of evolved objects as described earlier.

The first change occurs at Step 3(b)i. In this step, if an object is found at the location, the unevolved/evolved status of the object is first checked. This is easily accomplished by checking to see if the Evolve Pointer of the object points to itself or to some other address. If the Evolve Pointer of the object points to itself, indicating that the object is unevolved, then the algorithm can proceed as usual, copying the object to the ToSpace. However, if the Evolve Pointer points elsewhere, indicating that the

object is evolved, then another procedure must be followed. When an evolved object is encountered, the annex, instead of the original part, is copied into the ToSpace. The TIB Pointer of the annex will already be correct since the annex was created as an instance of the new class. Next, the Status Word found in the original part of the object is copied into the annex in the ToSpace. This ensures that locking and hash code state are preserved. The original part of the object in the FromSpace is overwritten with a forward to the annex in the ToSpace. Finally, the annex is scanned for object references to be added onto the end of the queue to be processed. At the end of this process, all that is left of the evolved object is the annex in the ToSpace, which is now in the unevolved format.

The second change concerns the Evolve Pointer of objects moved into the ToSpace. Regardless of whether the object moved into the ToSpace is unevolved or evolved (in which case it is the annex that is moved), the Evolve Pointer of the object is reset to point to itself. Since all objects moved into the ToSpace are now unevolved, this is consistent with the representation of unevolved objects.

Since the Semispace Copying GC algorithm traverses the entire set of live objects during a single cycle, it guarantees reconstitution of all evolved objects. With the reconstitution of all evolved objects, the operation of the Evolution JVM resumes to the Unevolved Stage.

Chapter 4

Test and Evaluation

This chapter details the tests used to validate the correct behavior and evaluate the performance of the Evolution JVM implemented as described in the last chapter.

Throughout this chapter, a radar tracking system example will be utilized to exercise the different aspects of the Evolution JVM. Figure 4-1 is a diagram of the classes that participate in this example.

As explained in Section 3.2.1, to test the Evolution JVM, code is first written in Java with `instanceof` expressions inserted where `evolveto` expressions are desired. This Java code is then compiled to bytecode using a standard Java compiler. The class files are then edited using a binary data editor to replace the *instanceof* opcodes with *evolve* opcodes. The resulting class files are then executed by the Evolution JVM.

4.1 Behavior

The following sections describe tests that ensure the proper behavior of the evolution capability within the Evolution JVM. These tests also verify the continued proper operation of other capabilities that might be affected by the modifications made to introduce evolution.

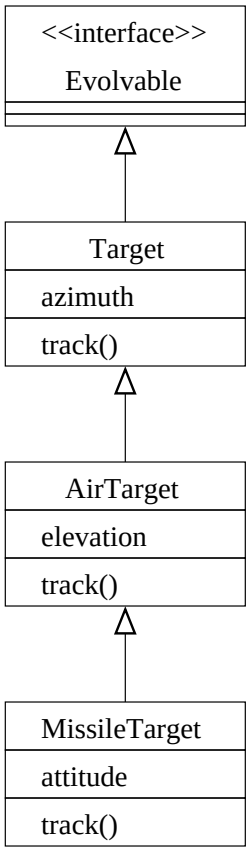

Figure 4-1: Target Class Hierarchy

4.1.1 Methods and Fields

Proper method invocation and field access are essential in the Evolution JVM, as its new conceptual features are revealed through these two pieces of functionality. To test these features, the `track()` methods of the Target, AirTarget, and MissileTarget were written to output the values of the fields of each class. This means that a Target instance outputs its azimuth, an AirTarget instance outputs its azimuth and elevation, and a MissileTarget instance outputs its azimuth, elevation, and attitude. At the start of the test, a Target is instantiated. The object's `track()` method is then invoked. The object evolves into an AirTarget. The `track()` method is invoked again. Finally, the object evolves into a MissileTarget, and the `track()` method is invoked once again. This test verifies both method invocation and field access as the class of the object changes.

This test revealed a low level detail overlooked in the first pass implementation of the Evolution JVM. As described in the Unevolved Stage Subsection of Section 3.2.5, all field accesses follow the Evolve Pointer. Therefore, it is essential that during the instantiation of a new object, the Evolve Pointer of the new object is initialized to point to itself. This initialization step was included in the portion of the JikesRVM that handles user program object creation. However, during bootstrapping, another part of the JikesRVM handles the writing of the boot image objects. Once the Evolve Pointer initialization procedure was added to the creation of boot image objects, the test completed successfully.

4.1.2 Recursive Methods

To ensure proper method invocation in an evolution environment, an additional test package was devised. This test sets out to verify proper operation during recursive method invocation. The classes are as in Section 4.1.1. However, for the Target and AirTarget classes, after outputting the fields of the class, the `track()` method includes an evolution expression operating on `this` moving the object to its direct subclass. After this evolution expression, a pseudo-recursive call to `track()` is made.

Although the `track()` invocations appear recursive, each call should actually invoke a different method due to evolution.

The Evolution JVM operates correctly under this test, invoking the three different `track()` methods as expected.

4.1.3 EvolveExceptions

This test package verified that EvolveExceptions were raised at the proper times. There are two conditions that cause an EvolveException to occur, according to the programmer interface described in Section 3.1. Both occur during the execution of the *evolve* opcode. First, if an evolution expression attempts to operate on null or a non-Evolvable object, an EvolveException should be raised. Second, if the subclass rule is violated, an EvolveException should be raised.

The method of testing these two conditions is obvious. The Evolution JVM correctly identified both conditions, raising the required exception.

4.1.4 Hash Code

JikesRVM incorporates hash code functionality into the object representation through the Status Word. The Evolution JVM inherits this approach, so a test package was created to verify proper behavior in an evolution environment. This test is almost identical to that described in Section 4.1.1. In addition to outputting the fields of the class, the `track()` methods output `this.hashCode()` as well. The hash code should remain constant through the evolutions.

The Evolution JVM operates correctly under this test, returning the identical hash code, even after evolution.

4.1.5 Locking / Multiple Threads

JikesRVM uses a subset of the bits of the Status Word as an index into an array of locks used for object locking. Because the Evolution JVM inherits this approach,

a test package was created to ensure proper locking operation before and after the evolution of an object.

In this test, the Target, AirTarget, and MissileTarget classes are as in Section 4.1.1. There are two threads, a main thread and a competing thread. The main thread starts first, instantiating a Target object. The main thread then locks the new object. At this point, the main thread starts the competing thread, which is trying to acquire the lock on the same object. The competing thread should block until the main thread releases the lock. However, in the main thread, the object evolves into an AirTarget and then into a MissileTarget, pausing after the evolutions to allow the competing thread to attempt to acquire the lock. The test verifies that the evolution expressions do not disturb the ownership of the lock by the main thread.

The Evolution JVM operates correctly under this test, excluding the competing thread until the main thread relinquishes the lock.

4.1.6 Garbage Collection Reconstitution

To test the reconstitution of evolved objects by GC, instrumentation code was added to the Evolution JVM for this purpose. The Semispace Copying GC algorithm traverses the entire set of live objects during its operation. For instrumentation, code was added to the GC to report any evolved objects encountered.

Again, the Target, AirTarget, and MissileTarget classes are as in Section 4.1.1. A Target object is instantiated. This object then evolves into an AirTarget. A GC is induced by creating a multitude of objects that are immediately discarded. During the GC, the evolved object is encountered, and the instrumentation code should report this. The GC should also reconstitute the evolved object as part of its evolution behavior. After this GC, another GC is induced using the same procedure. However, this time, the instrumentation code should not report any evolved objects. All evolved objects should have been reconstituted by the first GC.

The Evolution JVM operates correctly under this test, reconstituting the evolved object during the first GC.

4.1.7 Post Reconstitution

To further test the reconstitution operation, the Methods and Fields, Recursive Methods, Hash Code, and Locking tests were re-run with a GC induced after every evolution expression in the original test. All tests completed successfully except for the Hash Code test.

The Post Reconstitution Hash Code test revealed an important implementation detail that had been overlooked. JikesRVM uses address-based hash code generation. This approach is very simple until a GC moves an object. JikesRVM tackles this problem by expanding the object header of an object that is being moved and has previously produced its hash code. The hash code is stored in the expanded object header.

The Evolution JVM must address the case where an evolved object has previously produced a hash code. When this evolved object is reconstituted during GC, maintaining the hash code state becomes more complicated than just copying the Status Word, although the details are not particularly interesting. Once the proper method of maintaining the hash code state was added to the reconstitution procedure, the Post Reconstitution Hash Code test completed successfully.

4.2 Performance

The primary motivation behind the Evolution JVM is to improve the ability of Java implementations to model problem domain objects that exhibit class changing behavior. Although performance effects were considered in choosing a design, the desired evolution behavior was ranked before performance in comparing outcomes.

The tradeoff between performance and evolution behavior is most apparent in the implementation of field access. The Evolution JVM field access scheme is definitely less efficient than the original JikesRVM implementation. As described in the Unevolved Stage Subsection of Section 3.2.5, the Evolution JVM adds one additional IA32 instruction to the sequence of three instructions that normally serve as the translation of the *getfield* opcode to machine instructions. Worse yet, the extra

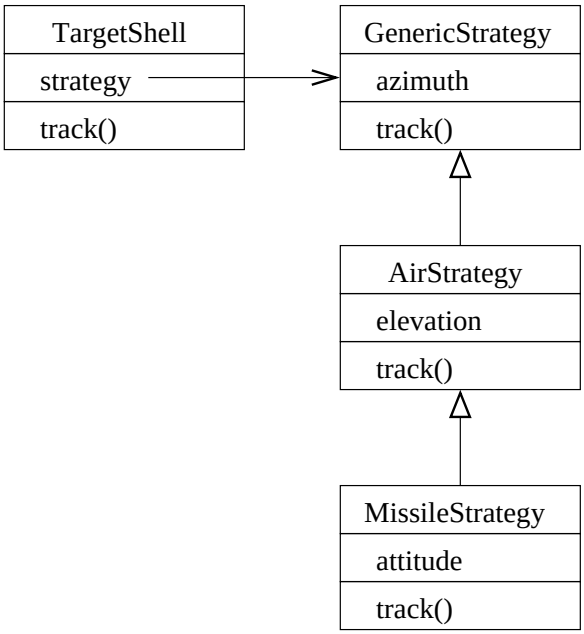

Figure 4-2: Implementation using Strategy Design Pattern

instruction is necessary regardless of whether the field being accessed is part of an unevolved or evolved object.

It is worth noting, however, that for the specific situations for which the Evolution JVM was designed, a performance advantage can be achieved by using evolution semantics and the Evolution JVM instead of the Java strategy design pattern and JikesRVM.

Consider the radar tracking system example described earlier and depicted in Figure 4-1. Imagine instantiating a Target object and then invoking its `track()` method. Figure 4-2 depicts an alternative implementation for accomplishing this same action using the strategy design pattern. Using this implementation, a TargetShell object is first instantiated. When the TargetShell object is instantiated, the `strategy` field is initialized with a newly instantiated GenericStrategy object. When the `track()` method of the TargetShell object is invoked, a field access is first performed to obtain the GenericStrategy object referred to by the `strategy` field. Finally, the `track()` method of the GenericStrategy object is invoked. The `track()` method of the GenericStrategy performs the same work the `track()` method of the Target class does in the Evolvable implementation. The strategy design pattern requires an additional

method invocation (invoking the TargetShell `track()`) and field access (accessing `strategy`) compared to the original implementation that relies on evolution.

In order to weigh the performance advantage of avoiding the strategy design pattern against the performance penalty of the Evolution JVM field access scheme, both the strategy and Evolvable approaches of inducing the track behavior were implemented and tested. In this test, the `track()` methods of the Target class (in the Evolvable approach) and GenericStrategy class (in the strategy approach) were simplified to only consist of a single field access. This simplification helps isolate the field access performance.

The strategy design pattern approach was executed on the JikesRVM while the evolution implementation was executed on the Evolution JVM. Both runs were on a RedHat 9 / AMD Athlon platform. To overcome a low level of resolution in the timing mechanism used to measure execution time, the two approaches were inserted into a loop. In the strategy design pattern loop, the `track()` method of a TargetShell object was invoked. In the Evolvable loop, the `track()` method of a Target object was invoked. The Evolvable approach on the Evolution JVM consistently outperforms the strategy design pattern approach on JikesRVM. This is expected due to the extra method invocation and field access involved in the strategy design pattern approach.

Because the `track()` methods of the Target and GenericStrategy classes consist of only a single field access, the Evolvable approach is expected to outperform the strategy design pattern approach. However, if additional field accesses are gradually added to the `track()` methods of the Target and GenericStrategy classes, it is also expected that the performance penalty of the field access scheme of the Evolution JVM will eventually erase the benefit of avoiding the strategy design pattern. This expectation was tested by doing just that. At ten field accesses per `track()` method, the Evolvable approach still achieves a significant advantage. At twenty accesses, the Evolvable / Evolution JVM approach is still superior, but the gap has been closed significantly. At thirty accesses per `track()` method, the strategy design pattern on JikesRVM achieves a better execution time, although evolution is still competitive. At forty accesses, the field access penalty starts to be significant, giving the strategy

design pattern on JikesRVM the clear advantage. This indicates that although the Evolution JVM can achieve a performance advantage by avoiding the strategy design pattern, high field access frequency can quickly erase this advantage.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis project set out to explore the usefulness, feasibility, implementation, and consequences of adding runtime class mutability to the Java language. The project was successful in that each of these aspects was examined in detail. However, the results of these examinations were both positive and negative. The next sections summarize the results.

5.1.1 Modeling Benefit

This thesis has shown, by example, that there are a variety of problem domains that exhibit class changing behavior. One category of these domains is characterized by objects that exhibit a life progression, like in the humans example. Another category of evolution-applicable domains is characterized by a refinement of information about the class of an object over time, like in the radar tracking system example or the computer network example.

By adding a Java capability to change the class of an object during runtime, the problem domains that exhibit class changing behavior can be implemented in a much more intuitive way. With this evolution capability, the operation of the Java implementation more closely models what is occurring in the problem domain. Using

the evolution capability is conceptually superior to the design pattern alternative; the design pattern approach does not map well to the problem domain and requires project-level invariants to be enforced.

The semantics for adding the evolution capability to Java are relatively simple, involving only the Evolvable interface, the new evolution expression, and the subclass rule that evolution expressions must satisfy. With the subclass rule, the evolution capability preserves type safety, a very important property to maintain for any new capability in Java.

Behavior tests performed to evaluate the Evolution JVM were successful, indicating that a Java evolution capability is feasible. The tests also indicate that there is no apparent impact on the standard Java behavior with the introduction of the evolution capability.

Therefore, from a modeling perspective, adding runtime class mutability to Java is quite beneficial, adding more representation power to the language with little costs in terms of semantic complexity or implementation complexity.

5.1.2 Performance Considerations

The modeling benefit of a Java evolution capability must be weighed against a possible performance degradation due to more complicated field access schemes.

In the implementation of the Evolution JVM presented in this thesis, there is a definite instruction count penalty for using the new field access scheme required for the evolution capability to work properly. Even if the problem domain is most amenable to the evolution capability, so that avoiding the strategy design pattern may have performance advantages (as described in Section 4.2), a high field access frequency could quickly erase any benefit.

Although the implementation presented here may sacrifice performance for the new evolution capability, this trade may not be absolutely necessary. Only an in depth examination of the performance issues can resolve the question of whether or not adding the evolution capability always results in a performance penalty. Such an examination should include a look at JVM optimizations currently in use and at

exploiting architecture details in implementing evolution. These topics are beyond the scope of this thesis.

5.2 Future Work

Although this thesis has skipped certain aspects of a complete Java evolution implementation on the basis of simplicity or obviousness, a complete implementation is necessary for a true evaluation. For instance, a functioning Java compiler that emits *evolve* opcodes would ensure that hidden complications were not overlooked. In such a Java compiler, inclusion of ideas like the `<evolve>` method could be explored.

As already mentioned, an investigation of field access schemes focusing on performance might demonstrate that a Java evolution capability can be implemented with little or no performance penalty. Taking advantage of architecture-specific performance details may be of particular value in improving performance. For example, in a highly pipelined architecture with branch prediction and a non-blocking memory system, it may actually be better to test whether an object is unevolved or evolved before following the Evolve Pointer. In this architecture, while the instruction to read the Evolve Pointer is waiting for an answer from the memory system, the branch predictor may have already predicted the branch associated with an unevolved object. If this is the case, then there is no need to wait for the Evolve Pointer read; the memory instruction to read an unevolved object field can be issued early. Compare this with the sequence implemented in the Evolution JVM in which the second memory access (to actually retrieve the field value) must wait for the value returned from the memory system for the Evolve Pointer even if the object is unevolved. Of course, any actual performance advantage might be dependent on specific architecture details like cache and branch prediction performance [5]. A study of these issues could yield a superior evolution field access scheme.

Bibliography

[1] Jikes RVM Online Documentation, 2004. `http://oss.software.ibm.com/developerworks/oss/jikesrvm/`.

[2] D. Scott Cyphers and David A. Moon. Optimizations in the Symbolics CLOS Implementation, 1990. Symbolics Inc., `http://www.apl.jhu.edu/~hall/text/Papers/CLOS-Optimizations.text`.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2003.

[6] Jikes RVM Project. *The Jikes Research Virtual Machine User's Guide*, 2004. Post 2.3.2.

[7] Richard Jones and Rafael Lins. *Garbage Collection*. John Wiley & Sons, 1996.

[8] Sonya E. Keene. *Object-Oriented Programming in Common LISP*. Addison-Wesley, 1989.

[9] Wilf R. LaLonde and John R. Pugh. *Inside Smalltalk*, volume one. Prentice-Hall, 1990.

[10] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[11] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[12] Mitchell Model. Comment on Runtime Type Mutability. Wiki comment, 2004. `http://c2.com/cgi/wiki?RuntimeTypeMutability`.